Fully Homomorphic Training and Inference on Binary Decision Tree and Random Forest*†

Hojune Shin‡ Jina Choi‡ Dain Lee Kyoungok Kim Younho Lee§

Department of Data Science, SeoulTech, Korea

Abstract

This paper introduces a new method for training decision trees and random forests using CKKS homomorphic encryption (HE) in cloud environments, enhancing data privacy from multiple sources. The innovative Homomorphic Binary Decision Tree (HBDT) method utilizes a modified Gini Impurity index (MGI) for node splitting in encrypted data scenarios. Notably, the proposed training approach operates in a single cloud security domain without the need for decryption, addressing key challenges in privacy-preserving machine learning. We also propose an efficient method for inference utilizing only addition for path evaluation even when both models and inputs are encrypted, achieving O(1) multiplicative depth. Experiments demonstrate that this method surpasses the previous study by Akavia et al.'s by at least 3.7 times in the speed of inference. The study also expands to privacy-preserving random forests, with GPU acceleration ensuring feasibly efficient performance in both training and inference.

Keywords: Decision Tree, CART, Privacy, CKKS, Fully Homomorphic Encryption.

1 Introduction

Decision trees excel in machine learning for their simplicity and ease of interpretation, effectively organizing data for regression and classification. Random Forests (RF), an evolution of decision trees, leverage multiple data subsets to build various trees, enhancing generalization and reducing overfitting, thus boosting effectiveness and reliability.

The focus of this paper is on training binary decision trees and random forests using CKKS homomorphic encryption (HE) in a outsourced cloud server environment. This approach is particularly crucial in maintaining the privacy and security of data when it is sourced from multiple institutions. By encrypting the data using HE, the study addresses the challenges of combining sensitive information from different sources without exposing it to potential security risks. The Homomorphic Binary Decision Tree (HBDT) method introduced in this paper employs a modified Gini Impurity index (MGI) for node splitting. Additionally, by utilizing the SIMD capabilities of CKKS, HBDT was able to calculate multiple instances of MGI simultaneously, and based on this, efficiently identify the split condition, enabling training without decryption. This modification speeds up the training process.

Moreover, this study showcases an efficient inference algorithm that operates on the outcomes obtained from the proposed training method. This algorithm, drawing parallels to the approach in Mahdavi et al's study [3], uses addition for tree evaluation to maintain a constant number of multiplication depths. A critical distinction of the proposed method is that it derives model data from encrypted sources, forming an encrypted model. This aspect is vital for efficient inference within encrypted environments, as it changes how inference inputs are handled and split conditions are

*A preliminary of version of this paper will appear at 29th European Symposium on Research in Computer Security (ESORICS) in Bydgoszcz, Poland, Sep. 2024.

†This work was supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) [NO.2022-0-01047, Development of statistical analysis algorithm and module using homomorphic encryption based on real number operation]

‡co-first authors, email:{rozn,cjina1102}@seoultech.ac.kr

§corresponding author. email:younholee@seoultech.ac.kr

processed at each tree node. This nuanced approach differentiates it from prior research, highlighting its comprehensive consideration of these aspects.

In situations where data from multiple institutions is combined for cloud-based training, the paper emphasizes the importance of preserving privacy through HE encryption. The risk of decryption during training poses significant concerns, such as potential data breaches by entities holding decryption keys and performance bottlenecks if a third party is involved. Additionally, due to the value of the data or legal regulations, decrypting original data, intermediate results, or models during or after training might be prohibited. *This study marks a pioneering achievement in creating an encrypted binary decision tree model. It uniquely processes encrypted training data using only HE operations within a single security domain of an outsourced server environment, effectively eliminating the dependence on multiple, non-collaborative servers.*

We also propose a method that allows efficient inference when both the model and input are encrypted in terms of the required wall clock time by reducing the amount of the required computation. The proposed method effectively utilizes the SIMD feature of CKKS to efficiently determine whether the input satisfies the split condition of each node. Moreover, we have achieved efficient inference by performing the tree evaluation with only O(1) multiplication, even when both the input and model are encrypted. Notably, in the same environments where only CPU is used for computation and with similar parameters, we show *the proposed method's inference performance was over 3.7 times higher than that in Akavia et al.'s study [2], which also protects the model using CKKS.*

This research also extends to efficient RFs for both training and inference, proposing new sampling techniques for training encrypted data and an efficient way to collect the inference results for many individual trees into a single ciphertext.

2 Backgrounds

2.1 Notation

The notations used are listed in Table 1. We assume the size of vectors used as inputs for encryption is always M. If the length of a vector included in a ciphertext is less than M, we assume zeros are appended to the end of that vector to make its size M.

2.2 Binary Decision Tree

We consider a binary decision tree (BDT) algorithm as CART [9]. We suppose the independent variables associated with the BDT are ordinal categorical variables represented as X_1, X_2, X_3,..., X_d, whose categories are represented as a set of positive integers $[1, n_i]$ $(i \in [1, d])$, respectively. Fig. 1 illustrates an example of a BDT and an inference with the BDT. We assume that the split conditions in non-leaf nodes are represented as $X_j \leq k$ $(k \in [1, n_j - 1])$ for each node N_i. During the inference stage, the path progresses to the left child node from a parent node if the split condition is satisfied; otherwise, it moves to the right child node.

As shown in the figure, if the input for inference is $(X_1,X_2,X_3) = (2,1,5)$, the result of evaluating the tree is $Y = 1$ where Y represents the target variable, which is expressed as a set of possible categories $[1, t]$. Additionally, in this paper, unlike conventional methods, during inference in the proposed HBDT, where both the model and the inference input are encrypted by CKKS HE, we represent the chosen edges selected in the inference process as 0, contributing to enhancing the inference speed. This approach can be used when the data is numeric by binning the data to make it a ordinal categorical variable.

RF utilizes multiple BDTs trained independently. Their predictions are aggregated to achieve a more accurate final prediction. Thus, RF can be treated as an ensemble model employing BTSs [22]. Section 6 discusses the extension of BDT to Homomoprhic RF.

2.3 CKKS (Cheon-Kim-Kim-Song) Scheme

CKKS is a homomorphic encryption algorithm that efficiently handles polynomial operations on encrypted real numbers and is widely applied in fields requiring real number computations for

Table 1: Notations and Conventions

Symbol	Meaning
$[a, b]$	The set of the integers between a and b, including the boundary.
d,t	d=# of independent variables, t=# of categories of target variable
n_i, n_{max}	n_i=# of categories in X_i, n_{max}=Max$(n_1, \cdots, n_d)$.
t_n, s_z	t_n= total # of rows in training data, $s_z = 2^{\lceil \log_2(t_n) \rceil}$
M, w	M=the total number of slots in a single ciphertext, $w = \lceil t_n/M \rceil$
X_i, Y	X_i=an independent variable ($i \in [1, d]$) and Y=the target variable
$x_i(y)$	a category in $X_i(Y)$
v	If $\vec{v}$ is a vector of real numbers, v is an encryption containing $\vec{v}$. If $\lvert\vec{v}\rvert > M$, it may represent multiple ciphertexts. The notation $\vec{v}$ is used interchangeably to represent the encryption of a vector $\vec{v}$.
$v[i]$	i-th slot of ciphertext $v, i \in [0, M-1]$
evk, sk, pk	evaluation key, secret key, public key
$\vec{1}^a(\vec{0}^a)$	A vector filled with as many 1s(0s) as a slots. If it is a vector with no a, this means that a is 1.
$(\vec{1}^a \lvert\lvert \vec{0}^b)^g$	A vector that is filled with a 1s and b 0s, starting at the front and repeating g times. The 1s and 0s can be reversed. $((a+b) * g \leq M)$
dep	The depth of the decision tree.
$\oplus, \ominus, \odot$	Add(), Sub(), Mult() operations (Refer to Section 2.3). $\odot$ is the hadamard product operator when used between plaintext vectors.
$\overrightarrow{x_{i,j}}, \overrightarrow{y_k}, c_{i,j,k}$	Refer to Section 5.
N_i	N_i is a node at position $i (\in N)$ in a tree. N_1 is the root node and N_{2i}, N_{2i+1} are left and right child of N_i, respectively.
Train_i	The training data used for training N_i.

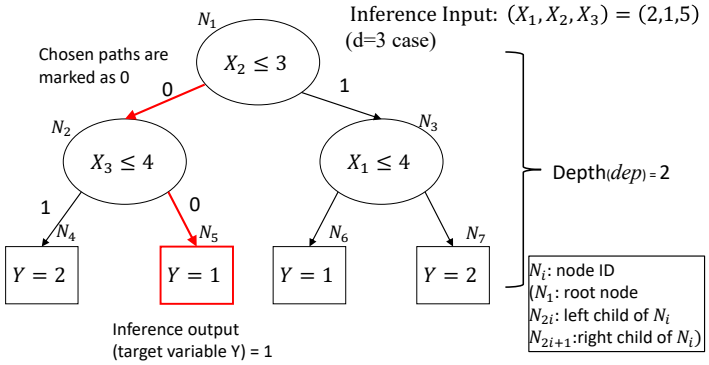

Figure 1: An example of a Binary Decision Tree and an Inference example

privacy-preserving machine learning [14]. It leaves the error for encryption thus, the computation result is approximate. CKKS supports the following algorithms given in Fig. 2 [27, 29].

The rescaling algorithm [14] is integrated into the Mult() algorithm. Following each multiplication and rotation, key switching occurs. For simplicity, detailed key-related information is omitted in this description. Our implementation utilizes both GPU versions of RNS-CKKS [24, 21, 11, 28] and the CPU version [17].

We use ApproxSign(x) [13, 17], which receives one ciphertext x and returns an encryption of a vector $(a_0, \cdots, a_{M-1})$ where $a_i = 1$ if $x[i] > 0$; or $a_i = -1$ if $x[i] < 0$ ($i \in [0, M-1]$). We also use the ApproxInv(x) [29, 17]: it returns an encryption of the multiplicative inverse of the values in x. It is implemented using the newton approximation method.

The CKKS parameters are as follows: M is 32768 for GPU and 16384 for CPU. The maximum modulus bit size of the ciphertext is 1555 for GPU and 777 for CPU, and the quantization bit size is 42 bits for GPU and 28 bits for CPU. The multiplication level provided after bootstrapping is 9 for GPU and 5 for CPU.

CKKS algorithms

- $\mathsf{KeyGen}(1^\lambda)$ takes a security parameter λ as input and returns pk, sk, and evk.
- $\mathsf{Enc}_{pk}(\overrightarrow{x})$ outputs x that maintains the vector structure as $\overrightarrow{x}$.
- $\mathsf{Dec}_{sk}(x)$ outputs $\overrightarrow{x}$ if x is a valid encryption from $\overrightarrow{x}$ Else it returns $\perp$.
- $\mathsf{Add}(x,y)(\mathsf{Sub}(x,y))$ outputs a new ciphertext $\overrightarrow{x}+\overrightarrow{y}$ ($\overrightarrow{x}-\overrightarrow{y}$). We use $\oplus(\ominus)$ to simplify the description. y can be replaced to a real number k in this case k is treated as a vector of M where every slot is set to k.
- $\mathsf{Level}(x)$ returns x's level, the number of further possible multiplication with x.
- $\mathsf{Mult}_{evk}(x,y)$ returns an (approximate) encryption of $(x_0 * y_0, \cdots, x_{M-1} * y_{M-1})$ whose level is $\mathsf{Min}(\mathsf{Level}(x), \mathsf{Level}(y)) - 1$. y can be replaced to a real number k in this case k is treated as a vector of M where every slot is set to k, and the output ciphertext's level is $\mathsf{Level}(x) - 1$. We use $\odot$ to simplify the description of this operation.
- $\mathsf{Rot}_{evk}(x,i)$ returns an encryption of $(x_i, x_{i+1}, \cdots, x_{M-1}, x_0, \cdots, x_{i-1})$, if $i \in [0, M-1]$. If i is negative, it refers to $M+i$. We use the notation '¡¡' to simplify the description of it.
- $\mathsf{Boot}_{evk}(x)$ returns a fresh ciphertext x' that has approximation of $\overrightarrow{x}$ if $\mathsf{Level}(x) \geq l_{min_{boot}}$, the number of required multiplication level to perform Boot. $l_{min_{boot}}$ depends on what bootstrapping algorithm is used and parameter.

Figure 2: CKKS algorithms

Table 2: Summary of Related Work ([1]: Inference privacy is out of scope. [2]: No cloud server exists in their settings.)

Requirements (refer to Sec. 4.2)	1)	2)	3)	4)	5)
[18, 19, 20, 1, 41, 42]	O	X	N/A[1]	X	N/A[2]
[33],[31]	O	O	O	X	X
[2]	O	O	O	X	O
[7], [8], [10], [4], [23], [26], [38], [43], [40], [35], [39], [45], [16], [5], [32], [3]	No training algorithm				

3 Related Work

Table 2 provides the summary of related work. The requirements 1) $\sim$ 5) are described in Section 4.2. A recent study comparable to our research is [2], which offers almost the same level of functionality as our proposed method, except for the use of decryption during the training process. We show that our HBDT has superior inference performance compared to [2]. Furthermore, the difference from the recent paper [3] is that, while [3] efficiently performs tree evaluation in situations where threshold and variable information are exposed, our HBDT provides efficient inference even when such information is encrypted.

There exist some research on utilizing multi-party computation (MPC) for privacy preserving tree-based models [34, 40, 44]. In their work, using MPC requires a lot of communication among multiple participants, and the communication complexity increases exponentially with the tree depth. Thus, it is difficult to be used when high network cost should be avoided or only poor communication is supported. However, the proposed approach utilizing homomorphic encryption performs all operations in ciphertext space, requiring minimal communication. Therefore, it is difficult to compare the performance between the proposed method and those with MPC. Therefore, in this study, the performance of the proposed method is compared only with privacy preserving decision tree training/inference methods that utilize homomorphic encryption with minimal communication.

4 Models

4.1 System Setting and Protocol Overview

Our system, based on configurations from existing literature (referenced as [29, 27]), is engineered to amalgamate training data across various security domains, aiming to develop superior quality models. In line with the legal standards of the authors' country, the system incorporates a Key

Manager (KM) tasked with scrutinizing inference results to ensure privacy compliance. The system's architecture involves three principal entities: Users, KM, and a Cloud Server (CS). Users, serving as data proprietors, contribute encrypted training data and also function as clients who receive inference outcomes from the CS. They provide encrypted input for inference, which is then processed by the CS. The CS is responsible for generating encrypted models using data from multiple users and conducting inferences using these models with the encrypted input supplied by users. Prior to dispatching these inference results to the users, they are thoroughly vetted by the KM to confirm legal compliance. An illustrative overview of all the participants and their operational protocols is provided in Fig. 3. During the key distribution phase in Fig. 3-(a), each user generates his/her own public and secret key pair. He/she sends his/her ID and public key to KM. KM sends each user the system public key(pk_s), after generating the system evaluation key (evk_s) and the public/secret key pair (pk_s, sk_s) for homomorphic encryption. Then, the system evaluation key (evk_s) and public key (pk_s) are sent to CS.

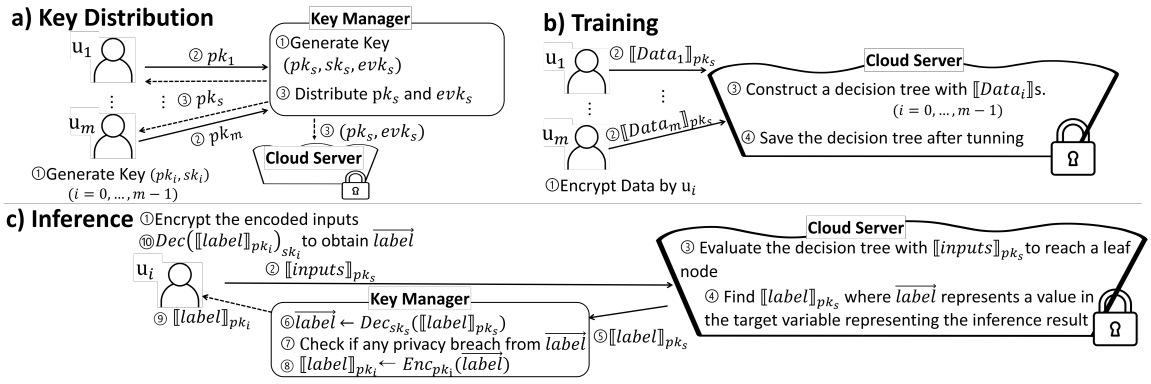

Figure 3: System Setting and Protocol Overview [27]

Security Model: Analogous to many previous studies, each participant can play the role of an adversary except KM, and their behavior is defined as an honest-but-curious (HBC) model. To avoid issues regarding the security of CKKS [30], we suppose that CKKS is CPA-secure [14, 12].

4.2 Problem Definition

We present the goals of this study, excluding efficiency issues. We designed HBDT to preserve the following requirements:

1. **Training data privacy**: The training data of a user should not be exposed to the other participants.
2. **Model privacy**: No participant should be able to obtain information about the model.
3. **Inference privacy**: The CS should not be able to obtain information about the inputs of the independent variables sent by users for inference.
4. **Non-interactive training**: Once the encrypted training data is passed to the CS, the subsequent training process is performed solely by the CS without the help of other entities.
5. **Single security domain for CS**: CSs might maliciously cooperate with each other because they exist in a single security domain, and no decryption keys are allowed to be used during training or inference, except for the final inference result.

5 Homomorphic Binary Decision Tree (HBDT)

Data representation: The format and representation of the training data and the input for inference are addressed. As both are generated as CKKS ciphertexts, they need to be representable in the ciphertext domain.

Fundamentally, all variables are represented using one-hot encoding in both training and inference. The length of each variable's vector representation is denoted by n_{max}. This allows for natural expansion by introducing additional vector elements containing 0 values. This representation enables expressing a single row of training data $(x_1, \cdots, x_d, y)$ as a vector containing $d \times n_{max} + t$ elements. We optimize efficiency by handling the training data on a column-wise basis. To illustrate, if the training data were in plaintext, each variable X_i would be represented as a set of vectors $(\overrightarrow{x_{i,1}}, \cdots, \overrightarrow{x_{i,n_{max}}})$. Here, $\overrightarrow{x_{i,j}}$ denotes the set of all j-th bits across the one-hot encoded values of X_i in the training data, with each containing t_n elements. Similarly, the data corresponding to Y as $(\overrightarrow{y_1}, \cdots, \overrightarrow{y_t})$, where $|\overrightarrow{y_j}| = t_n$. This is suitable when the users provide vertically partitioned training data to CS.

During the training process, all this data is transmitted to CS in an encrypted state. When encrypted, these variables are stored in ciphertexts in slots of M units each. Therefore, a column vector of a variable $\overrightarrow{x_{i,j}}$ is stored in a total of w ciphertexts, denoted as $c_{i,j,1}, \cdots, c_{i,j,w}$. The ciphertexts for $\overrightarrow{y_j}$ are represented as $c^{(y)}_{j,1}, \cdots, c^{(y)}_{j,w}$. Ultimately, the initial data used for training is defined as $\mathsf{Train}_1 = \{c_{i,j,k} | (i,j,k) \in [1,d] \times [1, n_{max}] \times [1,w]\} \bigcup \{c^{(y)}_{j,k} | (j,k) \in [1,t] \times [1,w]\}$.

Tree representation: In a BDT, as introduced in Section 2.2, non-leaf nodes maintain (j,k) to represent the splitting condition $X_j \leq k$, while leaf nodes have the inference result, denoted as $o (\in [1,t])$. However, in the proposed HBDT, the values j, k, and o are all maintained in an encrypted state. Moreover, even non-leaf nodes store the o value corresponding to the most frequent Y label present in that node during training. This is later used in special cases to determine the o value for leaf nodes. Another difference of HBDT from BDT is that to apply SIMD (Single Instruction, Multiple Data) for inference on encrypted inputs, the encrypted j, k, and o values for each node are collectively managed at the same level using the slots in the ciphertext, allowing for bulk storage and management. For more details, refer to Section 5.2.

5.1 HBDT-Training algorithm

The training algorithm in HBDT is depicted in the following Fig. 4. It first performs training N_1 with the initial data Train_1 then repeats training the descendant nodes until Queue is empty.

Training(Train_1,dep)

1. Queue.enqueue((N_1, Train_1))
2. While (Queue is not empty) do
2.1 $(N_i, \mathsf{Train}_i) \leftarrow$ Queue.dequeue()
2.2 $o, s \leftarrow \mathsf{FindMaxY}(\mathsf{Train}_i)$ %% most frequent Y label $= o$
2.3 $b \leftarrow \mathsf{CheckEmpty}(\mathsf{Train}_i, s)$%% $b = 0$ if Train_i has no row otherwise $b = 1$
2.4 If $(i == 1)$ then Set $N_i.y \leftarrow o$ else $N_i.y \leftarrow o * b + (1-b) * N_{\lfloor i/2 \rfloor}.y$
2.5 If $(i \geq 2^{dep})$ then go to Step 2 %% N_i is leaf node
2.6 $(N_i.j, N_i.k) \leftarrow \mathsf{FindSplitCond}(N_i, \mathsf{Train}_i)$ %% Representing condition $X_j \leq k$
2.7 $(N_{2i}, \mathsf{Train}_{2i}), (N_{2i+1}, \mathsf{Train}_{2i+1}) \leftarrow \mathsf{UpdateData}(N_i, \mathsf{Train}_i)$
2.8. Queue.enqueue($(N_{2i}, \mathsf{Train}_{2i}), (N_{2i+1}, \mathsf{Train}_{2i+1})$)
3. Return model $= \{N_1, \cdots, N_{2^{dep}-1}\}$

Figure 4: Training algorithm with encrypted data

In Step 2.2, FindMaxY() counts the rows corresponding to each Y label in Train_i to find the Y label with the highest frequency. If we assume the data is of plaintext, finding the Y label o with the largest $||\overrightarrow{y_o}||$ $(o \in [1,t])$ suffices. To execute this on encrypted data Train_i, FindMaxY() computes $c^{(sum)}_j \leftarrow \sum_{k=1}^{w} c^{(y)}_{j,k}$ for each Y label j, then uses $\log_2(M)$ rotations and addition operations on $c^{(sum)}_j$ to position the sum of values in all slots of $c^{(sum)}_j$ into the first slot. Combining the first slot values of all $c^{(sum)}_j (j \in [1,t])$ creates a single ciphertext c_{acc} where the combined values are spread across slots #1 to #t, followed by applying the FindMaxPosGroup() function (referenced in Appendix

A), which sets the slot with the highest value in c_{acc} to 1 and the rest to 0, then multiplies it to a ciphertext where an integer i is encoded in the ith slot. Then, we perform addition of every slot value and put the result in the 1st slot as before to obtain the resulting ciphertext o.

Additionally, it computes $c_{ty} \leftarrow \sum_{j=1}^{t} c_j^{(sum)}$ and creates an encrypted ciphertext s containing the sum of all slot values of c_{ty} in its first slot, following similar procedures. $s[0]$ has the number of valid labels in Train_i.

$\mathsf{CheckEmpty}()$ checks if there is a valid data row in Train_i by checking $s[0] > 0$ in Step 2.3. It performs $b \leftarrow s \cdot \mathsf{ApproxInv}(s)$ to return b. Based on the characteristics of the $\mathsf{ApproxInv}()$ in [29], if $s[0] = 0$, $b[0]$ becomes 0. If $s[0]$ is non-zero, $b[0]$ becomes 1.

We examine the pivotal step 2.6 of the training process. Unlike [2], this step does not require decryption at all. The goal of Step 2.6 is to find the split condition $X_j \leq k$ that minimizes the MGI at N_i. For every possible split condition $X_j \leq k$, we separate Train_i into Train_{2i} and Train_{2i+1}, calculating the frequency of each Y label value in both dataset. Viewing it as a plaintext, the rows represented by the one value in the vector $\overrightarrow{x_{j,k(\ell)}} \leftarrow \overrightarrow{x_{j,1}} + \cdots + \overrightarrow{x_{j,k}}$ corresponds to Train_{2i}, while $\overrightarrow{x_{j,k(r)}} \leftarrow \overrightarrow{x_{j,k+1}} + \cdots + \overrightarrow{x_{j,n_{max}}}$ corresponds to Train_{2i+1}, because the positions of ones in $\overrightarrow{x_{j,k(\ell)}}$ indicates the rows whose X_j value is at most k. Thus, the rows indicated by $\overrightarrow{x_{j,k(\ell)}}$ become the training data used by N_i's left child$(=N_{2i})$. Similar argument holds for $\overrightarrow{x_{j,k(r)}}$ as well.

Using this, we compute the frequency of each Y label q in Train_{2i} as $s_{j,k,q(\ell)} = \overrightarrow{x_{j,k(\ell)}} \cdot \overrightarrow{y_q}$ and calculate $Gini_{j,k(\ell)} = (\sum_{q=1}^{t} s_{j,k,q(\ell)})^2 - \sum_{q=1}^{t}(s_{j,k,q(\ell)})^2$ and $Gini_{j,k(r)}$ similarly as well. Finally, perform $Gini_{j,k} \leftarrow Gini_{j,k(\ell)} + Gini_{j,k(r)}$ to obtain the computation result. This value functions as a reference point for comparing the MGI calculated under the condition $X_j \leq k$ with the MGI from other split conditions, and it is directly proportional to the MGI value. For details of MGI, see Appendix B.

By conducting this process in an encrypted state, we calculate all $Gini_{j,k}$s for every $(j,k) \in [1,d] \times [1, n_{max}-1]$. For this, we create a pair of ciphertexts $c_{q(\ell)}$ and $c_{q(r)}$ where each pair of $s_{j,k,q(\ell)}$ and $s_{j,k,q(r)}$ are stored in the same slot position in ciphertexts $c_{q(\ell)}$ and $c_{q(r)}$, respectively. Then, we utilize SIMD to compute the $Gini_{j,k}$s of all variables and thresholds at the same time in parallel using $c_{q(\ell)}$ and $c_{q(r)}$. Upon performing the aforementioned process, we can obtain a ciphertext c_{Gini} as depicted in Fig. 5. After executing the $\mathsf{FindMinPos}()$ algorithm[1] on c_{Gini} and following the appropriate steps, we can obtain $N_i.j$ and $N_i.k$ in a format as the last part in Fig. 5.

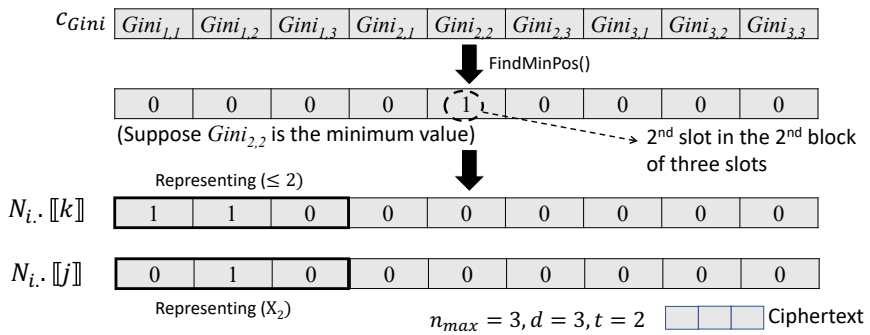

Figure 5: A part of $\mathsf{FindSplitCond}()$: after finding the MGIs of all possible cases.

We proceed to the $\mathsf{UpdateData}()$ function in Step 2.7 of Fig. 4. The plaintext version of this function can be implemented as shown in the Alg. 5.1. The key aspect lies in Step 4, in the creation of $\overrightarrow{x_{[1,k]}}$ (also $\overrightarrow{x_{[k+1,n_{max}]}}$ in Step 7). Subsequently, by performing component-wise multiplication on these vectors for all data, we generate the data to be used for training in the children nodes.

$\mathsf{UpdateData}()$ - plaintext version [1] $\mathsf{UpdateData} N_i, \mathsf{Train}_i$ $\{\overrightarrow{x_{p,q}} | (p,q) \in [1,d] \times [1, n_{max}]\}, \{\overrightarrow{y_p} | p \in [1,t]\} \leftarrow \mathsf{Train}_i, k \leftarrow N_i.k, j \leftarrow N_i.j$ $\overrightarrow{x_{[1,k]}} \leftarrow \sum_{i=1}^{k} \overrightarrow{x_{j,i}}$ $\overrightarrow{x_{p,q}} \leftarrow \overrightarrow{x_{p,q}} \odot \overrightarrow{x_{[1,k]}}$ for all $(p,q) \in [1,d] \times [1, n_{max}]$ $\overrightarrow{y_p} \leftarrow \overrightarrow{y_p} \odot \overrightarrow{x_{[1,k]}}$ for all $p \in [1,t]$ $\mathsf{Train}_{2i} \leftarrow \{\overrightarrow{x_{p,q}} | (p,q) \in [1,d] \times [1, n_{max}]\}, \{\overrightarrow{y_p} | p \in [1,t]\}$ Repeat Steps 4-5 with $\overrightarrow{x_{[k+1,n_{max}]}} \leftarrow \sum_{i=k+1}^{n_{max}} \overrightarrow{x_{j,i}}$ after replacing $\overrightarrow{x_{[1,k]}}$ to $\overrightarrow{x_{[k+1,n_{max}]}}$; then create Train_{2i+1} as Step 6. $(N_{2i}, \mathsf{Train}_{2i}), (N_{2i+1}, \mathsf{Train}_{2i+1})$

[1]This is $\mathsf{FindMinPosGroup}()$ where # of group is 1 and group size is M. It can be implemented easily with $\mathsf{FindMaxPosGroup}()$, explained in Appendix A.

The proposed training algorithm performs the above process using the encrypted Train_i, $N_i.j$, and $N_i.k$. The most challenging part lies in line 3 of Alg. 5.1, as it involves information encrypted representing j and k. To handle this, we utilize the following expression:

$$((N_i.j << r) \odot \vec{1}^1)[0] = 1 \text{ only if } j = r + 1 \quad (1)$$
$$((N_i.k << s) \odot \vec{1}^1)[0] = 1 \text{ only if } k > s (\geq 0) \quad (2)$$

We calculate $c'_r \leftarrow ((N_i.j << r) \odot \vec{1}^1)$ for every $r \in [0, d-1]$ as the equation (1), then copy the value of the first slot of c'_r to the rest of the slots in the same ciphertext. Then, we obtain $c'_{r,p,q} \leftarrow c'_r \odot c_{r,p,q}$ for every (r,p,q) $(\in [1,d] \times [1, n_{max}] \times [1,w])$. We eventually obtain only $c_{j,p,q} \leftarrow \sum_{r=1}^{d} c'_{r,p,q}$ as every $c'_{r,p,q}$ contains only zero values except when $r+1=j$. We suppose that we have $c_{j,p,q}$ for every pair of (p,q). Then, we create $c''_p \leftarrow ((N_i.k << p) \odot \vec{1}^1)$ for every $p \in [1, n_{max}]$ then copy the value in the first slot in c''_p to all the other slots as well. Then, we perform $c''_{j,p,q} \leftarrow c''_p \odot c_{j,p,q}$ for every $p \in [1, n_{max}]$ and make the summation of $c'''_{j,q} \leftarrow \sum_{p=1}^{n_{max}} c''_{j,p,q}$ for every $q \in [1,w]$. Then, we can obtain the desired outcome, $\overrightarrow{x_{[1,k]}} = \{c'''_{j,q}\}_{q=1,2,\cdots,w}$.

5.2 HBDT-Inference algorithm

For efficient inference, $N_i.j$ and $N_i.k$ for every N_i are further processed to be represented in $n_{max} \times d$ slots, as depicted in Fig. 6-(1)-(A). This representation allows us to check if the inference input being processed at node N_i satisfies the condition $X_j \leq k$ through the process shown in Fig. 6-(1)-(B).

The input for inference is also represented using one-hot encoding across $n_{max} \times d$ slots as shown in Fig. 6-(1)-(B). By performing operations in Fig. 6-(1)-(B), multiplying the model value with the input and moving the sum of all slot values to the first slot, regardless of where the inference result is, we can move the result into the first slot. By subtracting this value from 1, we can achieve the intended outcome as in Fig. 1, returning 0 when the split condition is satisfied and 1 otherwise when the path evaluation is performed in a tree.

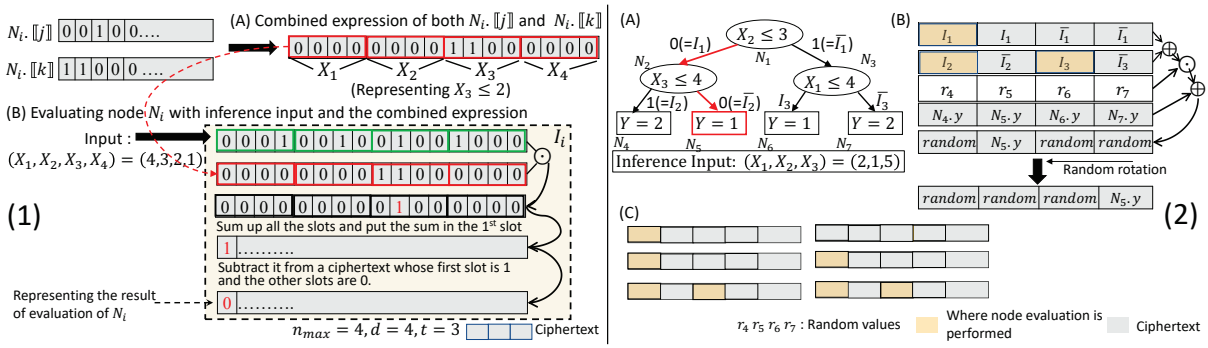

Figure 6: (1) Evaluation of a node with input in Inference (2) Path evaluation and block assignment in a ciphertext

The process for evaluating whether each node meets the split condition, as above, can be efficiently parallelized using SIMD capabilities. This is particularly effective when the number of available slots in the ciphertext, represented as M, is much greater than the size of the input, which is $n_{max} \times d$. To enable this, models for multiple nodes at the same level in the tree are placed in a single ciphertext. This is called the pre-processing step, and it is all conducted during the training process. Also, in the actual inference process, the inference input is replicated in such a way that its position aligns with these models, allowing for operations to be carried out on a slot-by-slot basis. The positions where the input is replicated is shown at 6-(2)-(C) and the description of the operation conducted is given in Fig. 6-(1)-(B).

Next, we discuss the method of evaluating the entire tree's path using the evaluation results for each node discussed above. Let's denote the evaluation process and its results at N_i as I_i, and

consider the complement of the bit in the result of I_i as $\overline{I_i}$. Additionally, let's refer to these individual I_i and $\overline{I_i}$ as blocks.

Fig. 6-(2)-(B) illustrates how the tree described in Fig. 6-(2)-(A) evaluates the red path in the proposed method. The yellow-colored blocks in Fig. 6-(2)-(B) indicate the places where the actual computations are performed to obtain the evaluation results in each of the nodes of the tree, while the rest are obtained by copying calculated results through rotation and addition operations. In Fig. 6-(2)-(B), blocks I_1, $\overline{I_2}$, and I_3 are computed as 0 while the rest are set to 1, because $X_2 \leq 3$ ($X_2 = 1$), $X_3 \not\leq 4$ ($X_3 = 5$), and $X_1 \leq 4$ ($X_1 = 2$). Therefore, adding the upper two ciphertexts of the blocks in Fig. 6-(2)-(B) results in only the block corresponding to the leaf nodes highlighted in red in Fig. 6-(2)-(A) having a value of 0, while the others have non zero values. After multiplying this result by a randomly sampled plaintext polynomial and adding to the ciphertext containing the target variable's label for each position corresponding to each leaf node (the fourth ciphertext in Fig. 6-(2)-(B)), only the label corresponding to the final leaf node along the red path of the tree ($N_5.y$ in Fig. 6-(2)-(B)) is stored in the respective slot position in the ciphertext, while the rest of the slots are filled with random values. Subsequently, to obfuscate the positions of the leaf nodes containing the resulting values, a random rotation is performed once, which produces the result of the inference.

In the proposed inference method, the client transmits her input of a single ciphertext where independent variables' one-hot encoded values are contained in the first $n_{max} \times d$ slots and nothing else. Thus, any information regarding the decision tree is not revealed to the client. Therefore the proposed method supports very high level of privacy regrading the model, which could not be achieved in the recent work so far [3].

For efficient inference process, CS needs to swiftly copy the inference input to the positions indicated by the yellow regions in Fig. 6-(2)-(B). To achieve this, we limit the number of blocks used in the ciphertext to $2^{\lceil \log_2(M/(n_{max} \times d)) \rceil}$ only. For instance, even if the ciphertext allows for five blocks, as depicted in Fig. 6-(2)-(C), we create and utilize the model with only the initial four blocks. This approach ensures fast alignment of the yellow blocks across multiple ciphertexts, as shown in the second and third ciphertexts of Fig. 6-(2)-(C), even if the model employs multiple ciphertexts.

Fig. 7 summarizes the overview of the proposed inference algorithm. model refers to ciphertexts that have undergone all the processes in Fig. 6-(1)-(A) and contain model values positioned only within the yellow blocks as in Fig. 6-(2)-(B). *dep* signifies the depth of the tree, determining the number of ciphertexts required to generate the model for a level of tree and the intervals between these yellow blocks within such model ciphertexts.

Inference(*input*,model,*dep*)

1. For the *input* containing input values in the first $n_{max} \times d$ slots, some rotation and addition operations are performed to generate ciphertexts aligning the positions of the input blocks with the yellow blocks in Fig. 6-(2)-(B).
2. Using the resulting ciphertexts from Step 1, some computations are executed at the positions of the yellow blocks as in Fig. 6-(2)-(B), computing the I_i values existing at those positions ($i \in N$) as in Fig. 6-(1)-(B).
3. Leveraging the outcomes from the yellow blocks, cryptographic operations involving appropriate copying and subtraction from constants are conducted for the remaining block positions of the ciphertexts in order to fill up the values I_i and $\bar{I}_i$ ($i \in [1, 2^{dep+1} - 1]$) in the rest of the blocks for tree evaluation, as in Fig. 6-(2)-(B).
4. The path evaluation is carried out as illustrated in Fig. 6-(2)-(B), and the resulting outcomes are transmitted to KM for further processing.

Figure 7: Inference algorithm with encrypted input

6 Extending to Homomoprhic Random Forests (HRF)

Training involves applying HBDT's process to numerous trees with two key differences. First, on the sampling over training data for each tree, to avoid heavy replacement sampling, it is performed selecting 63.2% of the data randomly and regarding 30% of these as chosen twice, resulting in rows selected twice having a value of 2 instead of 1 in the one-hot encoding. This ensures that rows are effectively sampled twice. This approximates the bootstrap sampling where the sampling size is the same as that of the training data. Second, for split condition setup at each node, only a subset of variables ($\lfloor\sqrt{d}\rfloor$) is considered, with large MGI values assigned to unsampled variables to prevent their selection.

Inference involves repeating the inference process of HBDT for multiple trees and consolidating results. The next step performs the majority voting among the inference results from multiple trees. However, majority voting over the encrypted data by HE requires significant computational resources. Therefore, the results from all trees are gathered into separate slots and transmitted to the user for manual majority voting. To address the challenge of transmitting results of several trees, an approach depicted in Fig. 8-(B) is followed. This involves using a function $f_{dep}()$ to compute the inference value and position it in a specific slot of the ciphertext. Also, by making the inference result using one-hot encoding, we can collect the inference results from multiple trees very quickly using only addition. In Fig. 8, if $N_5.y$ is one-hot encoded, after summing up all the results into a single ciphertext, no further operation including a rotation is necessary and just the resultant ciphertext is returned. This makes only one ciphertext needed to contain all trees' inference result.

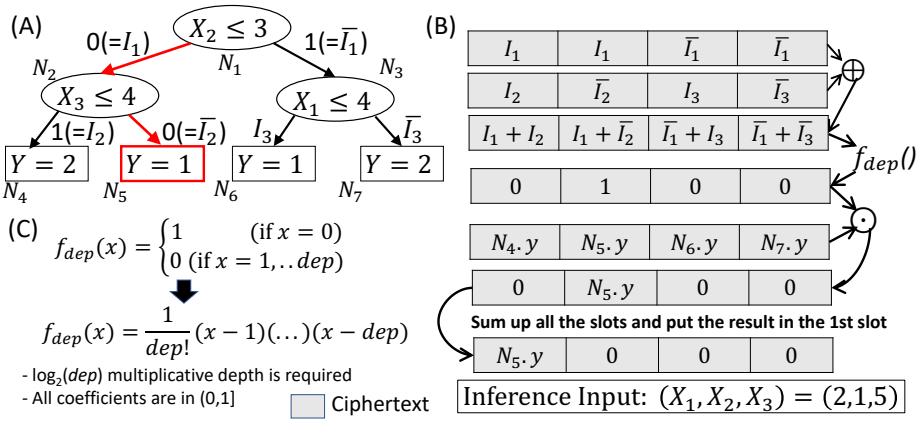

Figure 8: Path evaluation in a ciphertext for random forest

7 Experimental Results

We evaluated both GPU and CPU versions of our implementation in an AMD RYZEN 5950X CPU, NVIDIA Quadro RTX A6000 48GB GPU, and 128GB RAM environment. The results showcase the efficiency of training and inference performance, highlighting the outcomes from the version assisted by GPU to demonstrate the efficiency in both training and inference. Additionally, for comparing the inference algorithm with another study [2], we evaluated its performance using only one core of CPU, without GPU assistance.

For performance evaluation, we utilized the following datasets in Table 3. Scott refers to the binning method used to transform numerical variables into ordinal categorical variables [37]. Among these, the "sepsis" dataset was specifically intended to demonstrate the potential for training on large-scale data. All measurements were averaged over five training runs, ten inference runs, and a minimum of 33 CKKS operations per run to record the mean values.

Table 3: Dataset parameter $(t,d,n_{\max},t_n)$

iris (Scott) [36]	wine (Scott) [36]	cancer (Scott) [36]	sepsis [15]
3,4,9,100	3,13,11,119	2,30,18,380	2,3,101,110341

7.1 CKKS and Subroutines

Table 4 lists the performance of CKKS unit operations and subroutines. In the GPU version, The relative error of ApproxInv() is measured to $9.052E-04\%$. Other unit operations' relative errors are less than $1E-06\%$. The error attached to the plaintext due to encryption in the CPU and GPU versions are approximately $1.408E-02\%$ and $1.702E-06\%$, respectively, exhibiting a difference of about 8270 times. This discrepancy arises due to the different parameters used in the CPU and GPU versions of implementations, resulting in larger errors in the CPU version due to its shorter precision bit with smaller parameter values.

Table 4: Avg. Time (ms) of CKKS operations and subroutines (1st row: GPU, 2nd row: CPU)

Add	Mult(max. lv.)	Mult(min. lv)	Rot	Boot	ApproxSign	ApproxInv	SumGroup
0.017	0.72	0.26	0.60	149.0	1362	331	8.37
0.865	17.2	5.52	11.5	1403	16995	4513	189

Fig. 9 analyzes the training execution time. It describes the proportion of time taken for each step in Fig. 4, the time required for the model pre-processing stage as described in Fig. 6 for improving inference performance, and the total execution time in relation to depth. It is observed that the time taken is approximately double in proportion to the depth, and Step 2.6, which involves finding the split condition for nodes, takes the majority of time. Additionally, the pre-processing stage in Fig. 6 accounts for about 2 – 4% of the total execution time. As seen in the sepsis data, as the size of the data increases, the time taken to find the split condition and update the training data accordingly becomes a dominant factor.

We do not compare the required time for training between the proposed work and others because it is unfair. Prior works involve decryption during the training process. In [2], for instance, the client decrypts and computes the Gini impurity in the middle of the training process. However, our protocol performs the entire training process in CS with the encrypted training data. Thus, for example, the time taken to compute the homomorphic Gini impurity for Iris data with at the level 4 of the HBDT is approximately 117 seconds, whereas it only takes about 0.18 seconds to calculate the same one when they are of plaintexts.

Table 5 below displays the number of ciphertexts required for model storage. This refers to the case of the implementation on GPU. When the depth of the tree exceeds a certain number, it is observed that the number of required ciphertexts increases exponentially: the number of slots in the ciphertext required to store model information is proportional to the number of nodes at that level of the tree. Thus, as the number of nodes increases, more slots are required. Thus, if the level of three is over a certain threshold, the number of slots needed will exceed the capacity of a single ciphertext. As the level increases, the number of nodes doubles each time, leading to a doubling in the number of ciphertext slots required to store the model.

Table 5: The number of ciphertexts required to store model in training

Dataset	**iris (Scott)**								**wine (Scott)**								**cancer (Scott)**								**sepsis**			
Depth	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4
# of Ctxt	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	17	2	3	4	5	6	13	27	56	2	3	4	5

7.2 Performance of Inference

Fig. 10-(A) presents a comparison between the inference times of the proposed method and Akavia et al.'s approach [2][2]. To ensure fairness, the implementation of [2] used a 32-degree polynomial for the ApproxSign() function, while the proposed method utilized a 64-degree polynomial.

[2]https://github.com/intuit/Decision-Trees-over-FHE.git.

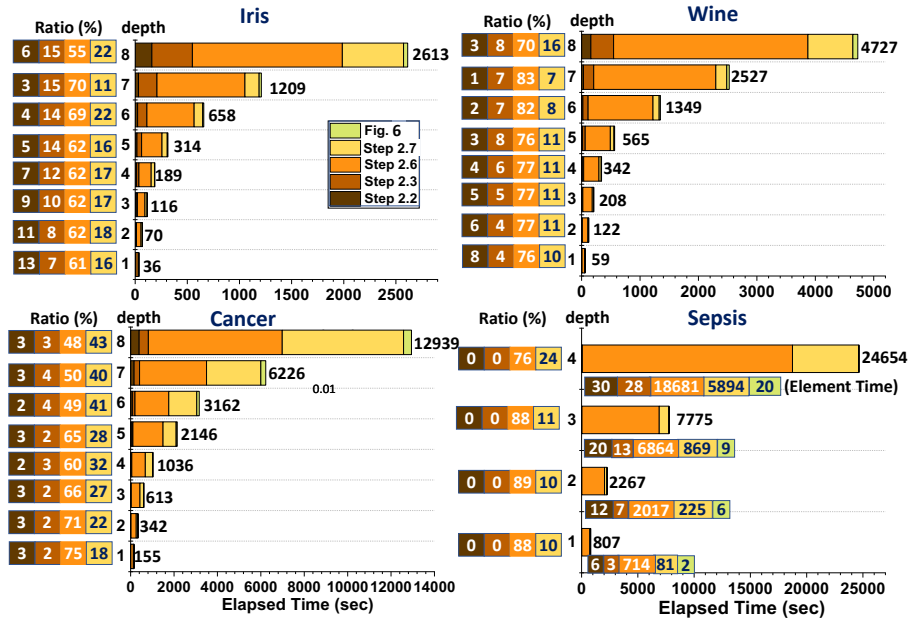

Figure 9: Execution time analysis in HBDT training

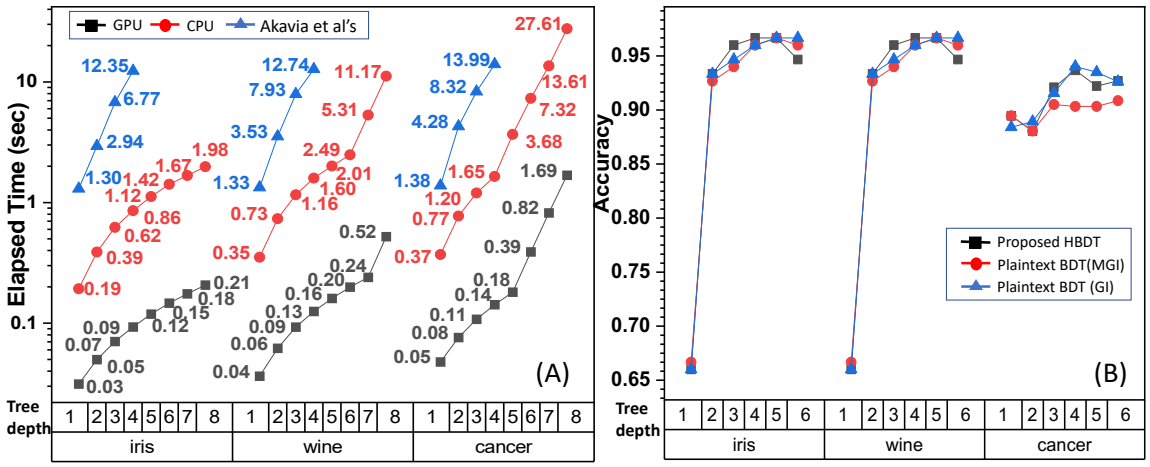

Figure 10: Inference: (A) execution time and (B) accuracy comparison

Additionally, both implementations used the ring of the polynomials whose maximum degrees are 32,768.

The results clearly demonstrate that the proposed method outperforms [2] across all datasets, being at least 3.7 times faster. In cases where the depth is 4 (with 16 leaf nodes), the proposed method showcases roughly eight times faster inference times. Fig. 10-(A) shows a graph comparing the inference execution times, showing that our inference times measured on CPU are at least 3.73 times (Cancer data, depth1) and up to 14.36 times (Iris data, depth4) faster than Akavia et al [2]. As a result, we can say that our CPU version implementation shows at least 3.7 times faster speed than Akavia et al.'s [2]. We evaluated the performance of Akavia et al.'s method [2] under the same computation environment as our CPU version. Furthermore, GPU-assisted inference times ranged from 0.03 seconds (depth 1, iris data) to 1.69 seconds (depth 8, cancer data). Notably, when using GPU acceleration, the inference performance was found to be less than 200 times slower compared to plaintext inference using a single CPU core for depths up to 4 in the same environment.

Fig. 10-(B) compares the accuracy of the inference results with the models created using the proposed method with encrypted data and the plaintext version of the data separately trained with

MGI and the standard Gini Impurity index (GI). As depicted in the graph, there is little difference in accuracy between the model trained using the proposed method and the plaintext-based model trained with GI, indicating comparable accuracy.

7.3 Performance of HRF

We evaluated the performance using 64 estimators in the random forest. The red lines in Fig. 11-(A) presents the training time of HRF. The black lines presents the comparison between the accuracy of the proposed method (HRF) and the plaintext version (RF). It's observed that the proposed method slightly lags behind but it is not noticeably high.

Fig. 11-(B) illustrates the inference performance achieved through GPU implementation. It demonstrates an inference time of approximately 6.5 – 8.5 seconds for all trees at a depth of 4. We can improve this with various methods, such as leveraging multiple GPUs. Please note that Steps 1 $\sim$ 3 depicted in Fig. 11-(B) align with Steps 1 $\sim$ 3 in Fig. 7. Step 4 begins from the execution of the $f_{dep}()$ function in Fig. 8-(B), and Step 5 involves assembling encrypted results from individual trees to generate the final encrypted output. Notably, Step 4 consumes much time as the evaluating f_{dep} for 64 trees require heavy computation.

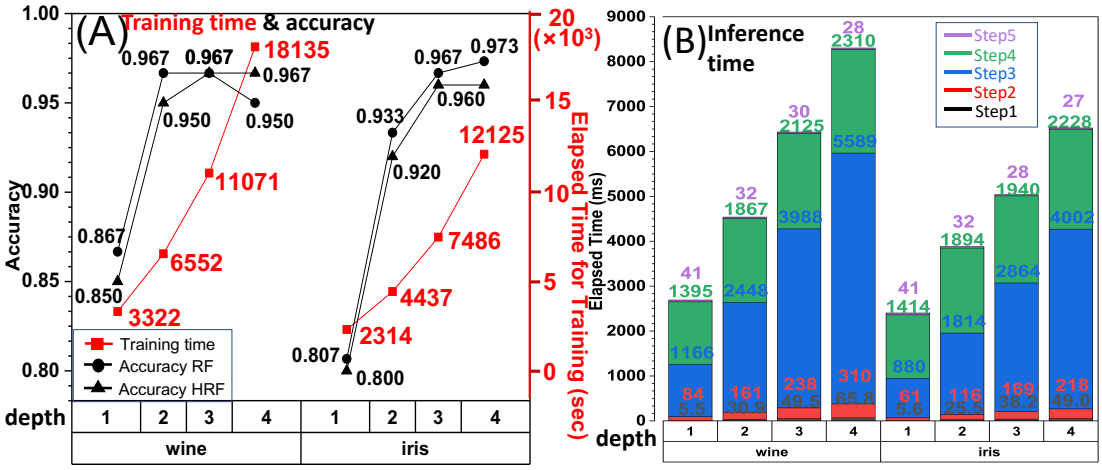

Figure 11: Performance analysis of Random Forest

8 Discussion

8.1 System Model without KM

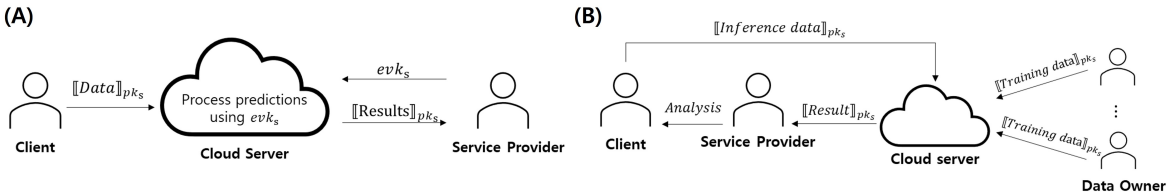

Figure 12: System model without KM Overview [25]

In this subsection, we discuss two system models that do not involve the use of KM. Before proceeding, it is important to clarify the term "Users"as employed in the proposed system model in Section 4.1. In the model, "Users" actually fulfill two roles: a data owner who supplies the training data to construct the machine learning models, and a service provider who offers the service to the

client. The proposed system model assumes that the inference result obtained by "Users" is relayed to their respective clients. However, this process is not explicitly depicted in the paper. To avoid confusion arising from the multiple meanings of the term "Users," we will refrain from using it in this section. Instead, we will employ the terms "data owner", "service provider," and "client" to clearly distinguish the different roles and entities involved in the system models.

The first model we introduce is from Kim et al's work [25], shown in Fig. 12-(A). This service supports safety by detecting emergency situations, such as falls of elderly people, in real-time, with preserving the privacy of the elderly people. For this, the client, who is an elderly person, sends a video encrypted with a system public key to the CS, which detects the elderly person's emergency situation through a machine learning model encrypted with the system evaluation key and notifies the service provider the detection result. The service provider decrypts the result with the system secret key and takes the necessary action for the client. In this model, CS can use the random forest model proposed in this paper.

The second is the model for tele-health services and personalized healthcare information analysis shown in Fig. 12-(B). In this model, the data owners provide training data encrypted with the system public key to CS, which generates a model through fusion and training. During the inference process, a client sends his/her encrypted data by system public key to CS, then it performs the inference with encrypted model and his/her encrypted data. CS sends the service provider the inference result. Finally, the service provider decrypts the result with the system secret key and explains the decryption result to the client.

The key feature of this protocol is that the service provider holds the system secret key, which allows it to decrypt the results returned from the CS and provide the explanation on the analysis result to the client. In situations where the data owner and service provider are different and there is only one service provider, KM is not needed.

Why is the problem hard without KM*:* the system model in Section 4.1 is similar to the second instance in that it includes multiple data owners, but differs in that every data owner also serves as a service provider. In this structure, every service provider must have the system secret key. When the system secret key is shared with multiple entities, malicious co-operation with CS can expose training data, as it becomes difficult to find out who gives the system decryption key to CS. This risk can be reduced in our system model due to KM.

8.2 Discussion on meeting the privacy requirements

This subsection discusses why the proposed method meets the privacy requirements provided in Section 4.2. The system setting in this paper considers the privacy of training data, models, inference inputs, and outputs. The training data and inference inputs are encrypted with the system public key and sent to CS, and CS cannot decrypt this information because it does not have the system secret key. This ensures that only the owner of the data knows his/her training data, and CS cannot access the actual content of the data during processing, thus maintaining the privacy of the training data (requirements #1, #3 in Section 4.2). Also, the training results, i.e. models, are encrypted with system public keys. The models are only used inside CS and are not exposed to the outside from CS, which protects the privacy of the models (requirements #2 in Section 4.2). This paper presents a system where a trusted KM exclusively manages the system secret key. Inference results are decrypted with the system secret key by KM and re-encrypted with the recipient's public key before being transmitted, which prevents access by third parties. (requirements #3 in Section 4.2) From the arguments above, we can see that the proposed method comprehensively ensures the privacy of training data, models, inference inputs, and results. In addition, the number of rounds of commmuinication between participants is minized in the proposed method. The training data is processed in a single unidirectional transfer from users to CS, and inference is also done in a single round of communication. This means the proposed method supports non-interactive training and inference, minimizing the communication cost of the system (requirements #4 in Section 4.2). Finally, by using a single CS to perform all processing, the model avoids multiple CSs and operates within a single security domain (requirements #5 in Section 4.2). Therefore, the proposed system satisfies all the requirements presented in Section 4.2, and effectively protects the privacy of training data, models, inference inputs, and results.

9 Conclusion

In this study, we introduced a novel HBDT (Homomorphic Encryption-based Decision Tree) and HRF (Homomorphic Encryption-based Random Forest) machine learning approach. We assessed their effectiveness through experiments using widely-used datasets. Previous research on privacy-preserving decision trees with homomorphic encryption faced challenges where certain aspects of the training process, such as calculating the Gini impurity, required decryption for computation, potentially exposing sensitive data. Our proposed method overcomes this limitation by enabling the generation of the machine learning model (HBDT) without the need for decryption. Furthermore, the inference process can be carried out without decryption until the encrypted inference result is obtained by the client-side (CS). Moreover, the inference process necessitates only $O(1)$ multiplication, allowing for efficient inference regardless of the tree's depth. Lastly, we extended this approach to develop a privacy-preserving random forest. We approximated replacement sampling for training data without decryption during the training phase. Additionally, by compressing the inference results from each tree in the forest, we achieved a reduction in inference time.

References

[1] Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: Proceedings of the 2000 ACM SIGMOD international conference on Management of data. pp. 439–450 (2000)

[2] Akavia, A., Leibovich, M., Resheff, Y.S., Ron, R., Shahar, M., Vald, M.: Privacy-preserving decision trees training and prediction. ACM Transactions on Privacy and Security **25**(3), 1–30 (2022)

[3] Akhavan Mahdavi, R., Ni, H., Linkov, D., Kerschbaum, F.: Level up: Private non-interactive decision tree evaluation using levelled homomorphic encryption. In: Proc. 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23). pp. 2945–2958 (2023)

[4] et al, D.C.: Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. IEEE Transactions on Dependable and Secure Computing **16**(2), 217–230 (2017)

[5] Azogagh, S., Delfour, V., Gambs, S., Killijian, M.O.: Probonite: Private one-branch-only non-interactive decision tree evaluation. In: Proc. 10th Workshop on Encrypted Computing Applied Homomorphic Cryptography (WAHC'22). p. 23–33. ACM (2022). https://doi.org/10.1145/3560827.3563377

[6] Bădulescu, L.A.: Experiments for a better gini index splitting criterion for data mining decision trees algorithms. In: 2020 24th International Conference on System Theory, Control and Computing (ICSTCC). pp. 208–212. IEEE (2020)

[7] Barni, M., Failla, P., Kolesnikov, V., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Secure evaluation of private linear branching programs with medical applications. In: Proc. 14th European Symposium on Research in Computer Security (ESORICS'09) #14. pp. 424–439. Springer (2009)

[8] Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine learning classification over encrypted data. In: NDSS Symposium 2015. p. 04_1_2. Internet Society (2015)

[9] Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and regression trees belmont. CA: Wadsworth International Group (1984)

[10] Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 498–507 (2007)

[11] Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full rns variant of approximate homomorphic encryption. In: International Conference on Selected Areas in Cryptography. pp. 347–368. Springer (2018)

[12] Cheon, J.H., Hong, S., Kim, D.: Remark on the security of ckks scheme in practice. Cryptology ePrint Archive (2020)

[13] Cheon, J.H., Kim, D., Kim, D.: Efficient homomorphic comparison methods with optimal complexity. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 221–256. Springer (2020)

[14] Cheon, J.H., et al.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology–ASIACRYPT 2017. pp. 409–437. Springer (2017)

[15] Chicco, D., Jurman, G.: Sepsis Survival Minimal Clinical Records. UCI Machine Learning Repository (2023), DOI: https://doi.org/10.24432/C53C8N

[16] Cong, K., Das, D., Park, J., Pereira, H.V.: Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In: Proc. 2022 ACM CCS. pp. 563–577 (2022)

[17] CryptoLab: HEAAN library (2022), https://heaan.it/

[18] De Hoogh, S., Schoenmakers, B., Chen, P., op den Akker, H.: Practical secure decision tree learning in a teletreatment application. In: Proc. FC 2014. pp. 179–194. Springer (2014)

[19] Du, W., Zhan, Z.: Building decision tree classifier on private data. In: Proceedings of the IEEE international conference on Privacy, security and data mining-Volume 14. pp. 1–8 (2002)

[20] Emekçi, F., Sahin, O.D., Agrawal, D., El Abbadi, A.: Privacy preserving decision tree learning over multiple parties. Data & Knowledge Engineering **63**(2), 348–361 (2007)

[21] Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Cryptographers' Track at the RSA Conference. pp. 364–390. Springer (2020)

[22] Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer (2009)

[23] Joye, M., Salehi, F.: Private yet efficient decision tree evaluation. In: Data and Applications Security and Privacy XXXII: 32nd Annual IFIP WG 11.3 Conference, DBSec 2018, Bergamo, Italy, July 16–18, 2018, Proceedings 32. pp. 243–259. Springer (2018)

[24] Jung, W., Kim, S., Ahn, J.H., Cheon, J.H., Lee, Y.: Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 114–148 (2021)

[25] Kim, M., Jiang, X., Lauter, K., Ismayilzada, E., Shams, S.: Secure human action recognition by encrypted neural network inference. Nature communications **13**(1), 4799 (2022)

[26] Kiss, Á., Naderpour, M., Liu, J., Asokan, N., Schneider, T.: Sok: Modular and efficient private decision tree evaluation. Proceedings on Privacy Enhancing Technologies **2019**(2), 187–208 (2019)

[27] Lee, D., Choi, J., Kim, K., Lee, Y.: Heaan-id3: Fully homomorphic privacy-preserving id3-decision trees using ckks. In submission (2023)

[28] Lee, J.W., Lee, E., Lee, Y., Kim, Y.S., No, J.S.: High-precision bootstrapping of rns-ckks homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In: Proc. Eurocrypt'21. pp. 618–647 (2021)

[29] Lee, Y., Seo, J., Nam, Y., Chae, J., Cheon, J.H.: Heaan-stat:a privacy-preserving statistical analysis toolkit for large-scale numerical, ordinal, and categorical data. IEEE Transactions on Dependable and Secure Computing (2023). https://doi.org/10.1109/TDSC.2023.3275649

[30] Li, B., Micciancio, D.: On the security of homomorphic encryption on approximate numbers. In: Proc. EUROCRYPT 2021: Part I 40. pp. 648–677. Springer (2021)

[31] Li, Y., Jiang, Z.L., Wang, X., Yiu, S.: Privacy-preserving id3 data mining over encrypted data in outsourced environments with multiple keys. In: IEEE Intl. Conf. on CSE and EUC. vol. 1, pp. 548–555 (2017)

[32] Liang, J., Qin, Z., Xue, L., Lin, X., Shen, X.: Efficient and privacy-preserving decision tree classification for health monitoring systems. IEEE Internet of Things Journal **8**(16), 12528–12539 (2021)

[33] Liu, L., Chen, R., Liu, X., Su, J., Qiao, L.: Towards practical privacy-preserving decision tree training and evaluation in the cloud. IEEE Transactions on Information Forensics and Security **15**, 2914–2929 (2020)

[34] Lu, W.j., Huang, Z., Zhang, Q., Wang, Y., Hong, C.: Squirrel: A scalable secure {Two-Party} computation framework for training gradient boosting decision tree. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6435–6451 (2023)

[35] Lu, W.j., Zhou, J.J., Sakuma, J.: Non-interactive and output expressive private comparison from homomorphic encryption. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 67–74 (2018)

[36] Markelle Kelly, Rachel Longjohn, K.N.: The uci machine learning repository (2023), https://archive.ics.uci.edu

[37] Scott, D.W.: On optimal and data-based histograms. Biometrika **66**(3), 605–610 (1979)

[38] Tai, R.K., Ma, J.P., Zhao, Y., Chow, S.S.: Privacy-preserving decision trees evaluation via linear functions. In: Proc. 22nd European Symposium on Research in Computer Security (ESORICS 2017) Part II 22. pp. 494–512. Springer (2017)

[39] Tueno, A., Boev, Y., Kerschbaum, F.: Non-interactive private decision tree evaluation. In: Data and Applications Security and Privacy XXXIV: 34th Annual IFIP WG 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25–26, 2020, Proceedings 34. pp. 174–194. Springer (2020)

[40] Tueno, A., Kerschbaum, F., Katzenbeisser, S.: Private evaluation of decision trees using sublinear cost. Proceedings on Privacy Enhancing Technologies **2019**(1), 266–286 (2019)

[41] Vaidya, J., Kantarcıoğlu, M., Clifton, C.: Privacy-preserving naive bayes classification. The VLDB Journal **17**(4), 879–898 (2008)

[42] Wang, K., Xu, Y., She, R., Yu, P.S.: Classification spanning private databases. In: Proceedings of the National Conference on Artificial Intelligence. vol. 21, p. 293. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999 (2006)

[43] Wu, D.J., Feng, T., Naehrig, M., Lauter, K.: Privately evaluating decision trees and random forests. Proceedings on Privacy Enhancing Technologies **4**, 335–355 (2016)

[44] Zheng, W., Deng, R., Chen, W., Popa, R.A., Panda, A., Stoica, I.: Cerebro: A platform for {Multi-Party} cryptographic collaborative learning. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2723–2740 (2021)

[45] Zheng, Y., Duan, H., Wang, C., Wang, R., Nepal, S.: Securely and efficiently outsourcing decision tree inference. IEEE Transactions on Dependable and Secure Computing **19**(3), 1841–1855 (2022)

A Appendix

A.1 Subroutines

This section introduces the subroutines used to build up the proposed method.

FindMaxGroup($input$,ns,gs,gn) returns a ciphertext where the first slot of each group has the maximum number among all the values in each group of gs slots in $input$. ns specifies the number of slots that have the values to be compared. Thus, $ns \leq gs$ and for an operational purpose, gs is even. gn specifies the number of groups in $input$. To speed up the operation, we suppose $gs \times gn \leq 2M/3$.

The number of invocation of ApproxSign() is $\lceil log_4(ns) \rceil$. The details of the description is given in Fig. 13. The process of the FindMax() function, responsible for finding the maximum value among

the data in each slot of a single ciphertext, is outlined in Fig. 13-(A). To minimize the number of ApproxSign() operations, the data is partitioned into four groups, A through D. Operations are performed within each group by comparing slots at identical positions to identify the slot with the highest value, keeping only that slot alive. This process continues until only one value remains. In Fig. 13-(B), the extended process of FindMaxGroup() is described. The crux lies in determining the maximum within each group. For instance, to identify the maximum value for two data groups, G1 and G2 (as detailed in Fig. 13-(C)), the data is distributed into four subgroups per group, as shown in the first illustration of Fig. 13-(B). After performing computations, the final illustration demonstrates the result, where only the highest values from the same slot positions among the subgroups within each group are retained. Repeating this process enables consolidating to only one ciphertext per group. Fig. 13-(C) illustrates the output of FindMaxGroupPos(). It takes as input a ciphertext containing data from multiple groups, marking slots where the maximum value within each group is located as 1, while setting the rest to 0. FindMaxGroupPos(c,ns,gs,gn) is executed as follows. First, it finds the biggest value among the values per each group in the input by executing FindMaxGroup(c,ns, gs, gn), and then creates a ciphertext (c_{max}) that fills all the slots solely with that the biggest value in each group. After that, c_{max} is subtracted from the input ciphertext, which is followed by adding a very small ϵ value to every slot in the subtraction result (c_{in}). Then, ApproxSign(c_{in}) function is executed to obtain c_{out}. Finally, we can obtain the final result by adding 1 then multiply 0.5 to every slot in c_{out}, where the slots of the biggest value are set to 1 in each group.

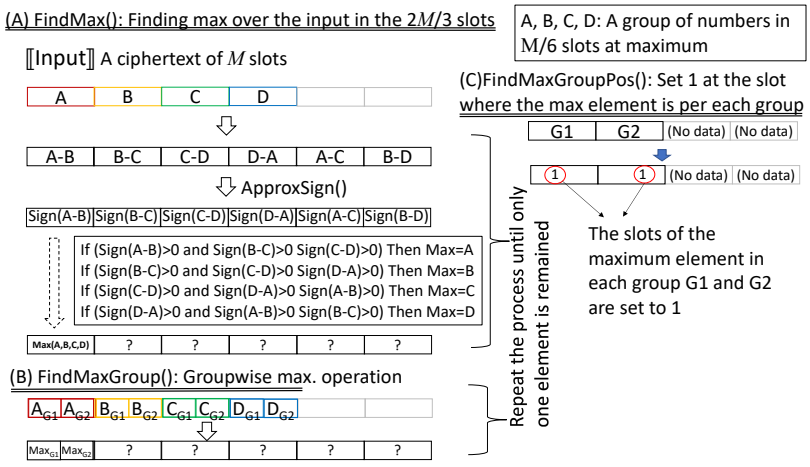

Figure 13: FindMaxGroup() and FindMaxGroupPos() algorithm

A.2 Modified Gini Impurity Index (MGI)

This study utilizes MGI to reduce the computation cost. MGI is a variation of the stadard Gini impurity [6], defined as follows:

$$
\begin{aligned}
MG(S|A) &= \sum_{t \in T} p(t)^2 G(t) = \sum_{t \in T} \frac{|S_t|^2}{|S|^2} \left(1 - \sum_{c \in C} \frac{|S_{t,c}|^2}{|S_t|^2}\right) \\
&= \frac{1}{|S|^2} \sum_{t \in T} (|S_t|^2 - \sum_{c \in C} |S_{t,c}|^2)
\end{aligned} \tag{3}
$$

where C represents the set of target classes in S and $p(\cdot)$ indicates the probability of the specific sample set in S (e.g., $p(i)$ denotes the probability of target class i, which can be defined as the proportion of the class i in S), $G(t)$ is the Gini impurity of samples at child node t, $|\cdot|$ denotes the cardinality of the set, T denotes the set of child nodes, S_t and $S_{t,c}$ represent the set of samples classified as child node t and the set of samples classified as child node t in class c. Because $|S|$ is

the common factor for all split rules, the best split rule based on the gain of the MGI is to minimize $|S_t|^2 - \sum_{c \in C} |S_{t,c}|^2$, which does not require division operations.